Ropes - Alternative String Representation

Leo Martel, Paul Martinez, Andy Moreland

June 3, 2014

1 Introduction

In this paper we will discuss the Rope data structure, a data structure intended to serve as a more robust and more performant alternative to the tradtional String type offered in most languages. The seminal paper regarding Ropes was written by Hans-J. Boehm, Russ Atkinson and Michael Plass in 1995. We will provide an overview of their paper, beginning with their justification for Ropes in the first place. We will continue with a more technical recap of the implmentation details and running time guarantees of a rope. As an exercise we have implemented our own Rope and we will give some rough benchmark estimates comparing various operations. Finally, in order to more easily show off the Rope data structure, we will discuss a visualization we created that allows one to actively interact with a Rope data structure and visualize its internal structure.

2 Justification for Expanded String Type

Boehm, Russ and Plass (BRP) begin their paper by discussing some of the faults of traditional string types and the various ways that they could be improved. We will provide a brief summary of their points.

Before attacking 'the traditional string type,' we must first define it. This traditional string type we refer to is the crude fixed length arrays of characters offered by languages such as C as Pascal. We can usually access individual characters via some sort of array access and perform higher level operations such as concatenation or substrings through library functions. Such implementations usually do not include any metadata along with the array (such as its length), and are usually mutable.

BRP list immutable strings as one of the characteristics of a more robust string type, noting that strings are often used to communicate between modules and thus one should be able to operate on a string without risk of modifying the original owner's copy of the string.

Additionaly, common string operations should be efficient. This is a natural desire, but BRP note that particularly string concatenation and substring operations should run fast and not require excessive space.

Perhaps most critically, a more robust string type should be able to scale and handle extremely long strings while remaining performant. BRP mention the original vi editor which was unable to handle large files due to a line length limit, and joke that a "six-month-old child randomly typing at a workstation would routinely crash some older UNIX kernels due to a buffer size limitation."

3 Implementation and Running Time

In order for string concatenation to run quickly, a desireable implementation should not copy the strings, which naturally suggests storing a string as a tree of smaller literals, which will be what we call our Rope. As defined by BRP, the leaves of this tree will be *flat* strings, the traditional raw character arrays, and internal nodes will represent the concatenation of their children. BRP note that because the leaf notes are immutable, a sequence of concatenation and substring calls could lead to many of these nodes being shared among multiple ropes, implying that the structure is techincally a directed acyclic graph. As they do, we will continue to refer to them as trees as we will be primarily considering only one rope at a time.

As a result of tree-storage structure and minor adjustments, we note that the following operations can easily be executed: arbitrary indexing, concatenation, substring and iteration. The first three all take logarithmic time through the use of self-balancing trees and the third takes linear time. Though, due to immutability, tree-rebalances are more expensive due to the copying of concatenation nodes. Thus BRP recommended that tree-rebalances are done rarely and only explicitly. To keep the size of the rope from getting out of hand in the meatime, BRP recommedn the follwing approach:

3.1 Concatenation

A common use case for strings is continaully concatenation at one end. Thus whenever the right argument is a string literal, we would like to avoid growing the tree excessively. We can do this by accounting for two cases. One, if both arguments are string literals, we will simply combine them into a new string literal. Two, if the left argument is a concatenation node but its right child is a string literal, then we will combine the right child with the right argument to form a new string literal and then return a new concatenation node with the left child. In other cases we simply create a new concatenation node. These two rules make the case of continually appending to the end of a string very efficient.

3.2 Substring

When taking substrings

4 Benchmarking

We wrote some code.

5 Visualization

Text editors, wooo.